Implementation of ARTiS, an Asymmetric Real-Time Extension of SMP Linux

Philippe Marquet, Éric Piel, Julien Soula, Jean-Luc Dekeyser

Implementation of ARTiS, an asymmetric real-time extension of SMP Linux*

Philippe MARQUET, Éric PIEL, Julien SOULA, and Jean-Luc DEKEYSER
Laboratoire d'informatique fondamentale de Lille
Université des sciences et technologies de Lille
France
{Firstname.Lastname}@lifl.fr

Abstract

ARTiS is a real-time extension of GNU/Linux dedicated to SMP systems (Symmetric Multi-Processors). ARTiS divides the CPUs of an SMP system into two sets: real-time CPUs and non real-time CPUs. Real-time CPUs execute preemptible code only, thus tasks running on these processors perform predictably. If a task wants to enter into a non-preemptible section of code on a real-time processor, ARTiS will automatically migrate this task to a non real-time processor. Furthermore, dedicated load-balancing strategies allow all the system's CPUs to be fully exploited.

The purpose of this paper is to describe the basic API that has been specified to deploy real-time applications, and to present the current implementation of the ARTiS model, which was achieved through modifications of the 2.6 Linux kernel. The implementation is build around an automatic migration of tasks between real-time and non real-time processors and the use of a load-balancer. The basic function of those mechanisms is to move a task structure from one processor to another. A strong constraint of the implementation is the impossibility for the code running on an RT processor to share a lock or to wait after another processor.

1 Real-Time Multiprocessing

In the framework of the HYADES project [5], we have identified the need for an operating system providing real-time capabilities mixed with high performance. Given this specifications, the use of an operating system such as Linux in conjunction with a commodity symmetrical multiprocessor system is a good candidate: SMPs are recognized for their ease of programming while the support of SMP systems in Linux is now mature.

Nevertheless, the Linux operating system is a general purpose system. Even if it provides all the services a programmer may wish, there are problems when dealing with real-time aspects. Even the last version of the so called "preemptible Linux kernel" only targets soft real-time applications and may show latencies of up to 40ms (according to our measurements presented in [6]).

An approach which combines real-time requirements with the basic nature of the SMP system is the asymmetric multiprocessor principle. On an SMP system one or several processors are dedicated to real-time tasks: a spatial reservation of resources replaces the usual temporal reservation of resources. Any processing that may jeopardize the real-time constraints of the applications is contained in those processors which are not dedicated to real-time. Despite the fact that this approach is practicable [2, 9], it has a major drawback: real-time processors may be idle while non real-time processors are overloaded. The static separation between real-time processors and non real-time processors leads to resource wasting.

ARTiS is an evolution of asymmetric multiprocessing. It allows a task to run on real-time CPUs as long as it stays in preemptible mode: when needed for real-time processing, the processor is able to interrupt the task without delay. ARTiS is based on the automatic migration of a task from a real-time processor to a non real-time processor before it enters into a non preemptible section of code, this ensures

*This work is partially supported by the ITEA project 01010, HYADES

real-time reservations on the real-time processors.

The rest of the paper is organized into four parts. The ARTiS model is presented in the next section. The current API suggested for the deployment of real-time applications on ARTiS is then summarized. The main points of the ARTiS implementation as a modification of the Linux kernel are explained in the following section. The last section lists the future developments we plan for the ARTiS project.

2 ARTiS Model

ARTiS promotes a programming model based on a user-space programming of real-time tasks: the programmer uses the usual POSIX and/or Linux API to write his applications. These tasks are real-time in the sense that they are identified as high priority and are not perturbed by any non real-time activities. For these tasks, ARTiS targets a maximum response time below 300μs.

The core of the ARTiS solution is based on a strong distinction between real-time and non real-time processors, and also, on migrating tasks which attempt to disable the preemption on a real-time processor. To provide this system ARTiS proposes:

- The partition of the processors into two sets,
- Two classes of RT processes,
- A specific migration mechanism,
- An efficient load-balancing policy,
- Asymmetric communication mechanisms.

2.1 Processor Partitioning

The processors of the SMP system are partitioned into two sets: an NRT CPU set (Non Real-Time) and an RT CPU set (Real-Time). Each one has a specific scheduling policy. The purpose is to insure the best interrupt latency for particular processes running inside the RT CPU set.

2.2 Process Classes

Two classes of RT processes are identified. These are standard RT Linux processes, they just differ in their mapping:

- Each RT CPU has just one bound RT Linux task, called **RT0** (a real-time task of highest priority). Each of these tasks has the guarantee that its RT CPU will stay entirely available to it. Only these user tasks are allowed to become non-preemptible on their corresponding RT CPU. This property ensures the lowest possible latency for all RT0 tasks. The RT0 tasks are the hard real-time tasks of ARTiS. Execution of more than one RT0 task on a given RT CPU is possible but in this case it is up to the developer to verify the feasibility of such a scheduling.

- Each RT CPU can run other RT Linux tasks but **only** in a preemptible state. These tasks are called **RT1+** (real-time tasks of priority 1 and below). They can use CPU resources efficiently if RT0 tasks do not consume all the CPU time. To keep a low latency for RT0 tasks, the RT1+ processes are automatically migrated to an NRT CPU by the ARTiS scheduler when they are about to become non-preemptible. The RT1+ tasks are the soft real-time tasks of ARTiS. They have no firm guarantees, but their requirements are taken into account by a best effort policy. They are also the main support of the intensive processing parts of the targeted applications.

- The other, non real-time, tasks are named "Linux tasks" in the ARTiS terminology. They are not related to any real-time requirements. They can coexist with real-time tasks and are eligible for selection by the scheduler as long as the real-time tasks do not require the CPU. As in the case of the RT1+ tasks, the Linux tasks will automatically migrate away from an RT CPU if they try to enter into a non-preemptible code section on such a CPU.

- The NRT CPUs mainly run Linux tasks. They also run RT1+ tasks when these are in a non-preemptible state. To insure the load-balancing of the system, all these tasks can migrate to an RT CPU but only in a preemptible state. When an RT1+ task runs on an NRT CPU, it keeps its high priority above the Linux tasks.

2.3 Automatic Migration

A specific migration mechanism aims at insuring the low latency of the RT0 tasks. All the RT1+ and Linux tasks running on an RT CPU are automatically migrated toward an NRT CPU when they try to disable the preemption. One of the main changes required to the original Linux load-balancing mechanism is the removal of inter-CPU locks. To effectively migrate the tasks, an NRT CPU and an RT CPU have to communicate via queues. This migration is implemented thanks to a lock-free FIFO to

avoid any active wait by the ARTiS scheduler based on [10]. Section 4.1 details this mechanism.

2.4 Load-Balancing

An efficient load-balancing policy allows the full power of a SMP machine to be exploited. Usually a load-balancing mechanism aims to move the running tasks across CPUs in order to insure that no CPU is idle while tasks are waiting to be scheduled. Our case is more complicated because of the specificities of the ARTiS tasks. By definition, the RT0 tasks will never migrate. The RT1+ tasks should migrate back to RT CPUs quicker than Linux tasks: the RT CPUs offer latency warranties that the NRT CPUs do not. To minimize the latency on RT CPUs and to provide the best performance for the global system, specific asymmetric load-balancing algorithms have been defined [8]. Section 4.2 outlines these algorithm implementation.

2.5 Asymmetric Communications

Asymmetric inter-process communication mechanisms are provided. On SMP machines, tasks exchange data by read/write mechanisms on the shared memory. To insure coherence, critical sections are needed. These critical sections are protected from simultaneous concurrent access by lock/unlock mechanisms. This communication scheme is not suited to our particular case: an exchange of data between an RT0 task and an RT1+ task will involve the migration of the RT1+ task before this latter takes the lock, in order to avoid entering into a non-preemptible state on an RT CPU. Therefore, an asymmetric communication pattern should use lock free FIFO in a one-reader/one-writer context. These communication mechanisms are not yet definitively defined and form the main part of our future projects.

3 Real-Time API

A basic ARTiS API has been specified. It allows the deployment of applications on the current implementation of the ARTiS model, available as a modification of the 2.6 Linux kernel. ARTiS applications are defined by a configuration of CPUs, an identification of real-time tasks and their processor affinity.

3.1 Machine Partitioning

The CPUs are partitioned into two sets, the RT and NRT CPU, via a basic `/proc` interface :

- A value greater than or equal to 1 in `/proc/artis/activate` dynamically activates the ARTiS functioning,
- The `/proc/artis/maskrt` file contains the mask of the RT CPUs. It can be modified while ARTiS is inactive.

The only limitation is to keep at least one CPU identified as an NRT CPU. We are also investigating the use of the CPUSETS patch provided by Bull that allows such partitioning of a multiprocessor [3].

To maintain coherence with this machine partitioning, a redirection of the interrupts has to be programmed. All IRQs must be delivered exclusively on the NRT CPU, excepting those IRQs used by the RT0 tasks, which must be delivered on the CPU hosting the task. The `/proc/irq/*/smp_affinity` files are used for this purpose.

3.2 RT Process Identification

The RT0 ARTiS tasks are identified as Linux tasks scheduled with the FIFO scheduling policy (`SCHED_FIFO`) and having the highest priority. The POSIX functions `sched_setscheduler()`, `sched_setparam()` and `sched_get_priority_max()` are used for this purpose. An RT0 task must be bound to an RT CPU. The non POSIX `sched_setaffinity()` primitive is used for this. Obviously, the set of CPUs on which an RT0 is allowed to run on must be limited to a single CPU, and this CPU must be an RT CPU.

Figure 1 presents an outline of the code a task may include in order to be identified as an RT0 task. ARTiS also comes with a basic library that provides functions to register and unregister an RT0 task:

```
int artis_enter_rt0 (pid_t pid, int rt_cpu);
int artis_leave_rt0 (pid_t pid);
```

The RT1+ tasks are all the Linux tasks associated with either the FIFO or round-robin scheduling policy (`SCHED_FIFO` or `SCHED_RR`). As with the standard POSIX definition, the priorities of these tasks define their relative priority. The ARTiS library provides the following two functions to identify these tasks:

```
int artis_enter_rt1plus(pid_t pid,
              int policy, int priority);
int artis_leave_rt1plus(pid_t pid);
```

The so-called Linux tasks, *i.e.* the non real-time tasks, are all tasks scheduled with the usual Linux `SCHED_OTHER` policy.

The CPU affinities of non RT0 tasks must include an NRT CPU, otherwise they will no longer

```
unsigned int rt_cpu;
struct sched_param schedp;

/* lock the address space of the process */
if (mlockall(MCL_CURRENT|MCL_FUTURE) != 0)
        perror(...);

/* set the scheduling policy */
memset(&schedp, 0, sizeof(struct sched_param));
schedp.sched_priority = sched_get_priority_max(SCHED_FIFO);

if (sched_setscheduler(0, SCHED_FIFO, &schedp) != 0)
        perror(...);

/* bound the process to the rt_cpu CPU */
if (sched_setaffinity(0, sizeof(unsigned long), 0x1UL << rt_cpu ) == -1)
        perror(...);
```

FIGURE 1: *RT0 identification*

be eligible for execution when entering a non-preemptible code section. In addition, the CPU affinities of RT1+ tasks should also include at least one RT CPU.

4 ARTiS Implementation

The ARTiS model is currently implemented as a modification of the 2.6 Linux kernel. The main modification concerns the automatic migration of a non RT0 task that is about to enter into a non preemptible section of code on an RT processor: this is a requirement from the ARTiS model. Furthermore, to benefit of the whole system, tasks must be a able to move from one processor to another depending on the processor load. The usual algorithm included in Linux for this purpose has also been enhanced to deal with the real-time aspects of ARTiS.

4.1 ARTiS Migration

ARTiS migration refers to the mechanism that automatically migrates a task from an RT processor to an NRT processor because the task is about to enter a non preemptible section of code. As such, the mechanism requires that firstly the point of entry into such a section of code be identified, and secondly that the task be moved from an RT processor to an NRT processor. This latter relies on a specific implementation which guarantees that an RT processor will not wait for a lock shared with an NRT processor.

4.1.1 Migration Triggering

The ARTiS automatic migration mechanism is not systematic. Many conditions must be satisfied before allowing the migration: automatic migration only effects RT processors, neither RT0 tasks, nor the idle task are concerned, and interrupt handlers are not considered.

Migration must be triggered as soon as a task enters into a state where it will not be able to stop, so that an RT0 task can be scheduled. We have identified two paths of this kind:

- the preemption is disabled (IRQs are still handled but no re-scheduling is possible), i.e. a call to `preempt_disable()`,
- the interruption is disabled (IRQs are no longer received), i.e. a call to `local_irq_disable()`.

A task that enters into one of these two functions must migrate to an NRT processor. These two functions have been patched to include a call to the function `artis_try_to_migrate()`.

This function checks the migration conditions and, if the migration is possible, effectively triggers the migration by calling `artis_request_for_migration()`.

Moreover, one can locally disable the migration in order to protect a part of the code, for instance, the `schedule()` function. To achieve this, ARTiS provides the two functions `artis_migration_disable()` and `artis_migration_enable()`. They (un)set the so-called ARTiS flag that is used as a complement to validate an automatic migration.

4.1.2 Task Migration Pathway

Inter-CPU locks are unsafe because an NRT processor may block an RT processor that shares the lock, consequently ARTiS is not able to directly use the original Linux run-queues. The RT processor must not take the lock on the local run-queue and the lock on the destination run-queue at the same time.

We take advantage of the fact that the scheduler already takes a lock on its run-queue in order to perform migration during the scheduler execution. Therefore, the actions of dequeuing and queuing are executed by different CPUs, the link between them being achieved by an intermediate queue specific to ARTiS, called RT-FIFO. On ARTiS, an RT-FIFO connects every RT processor to every NRT processor.

A task triggers its own migration using a call to `artis_request_for_migration()` but a task can not queue itself in an other run-queue because, in this case, it would be runnable on two CPUs at the same time. Consequently, it needs a helper task in the same way that changing its own processor affinity requires the `kmigration` kernel thread. In ARTiS, the duty of helper task is devolved to the next scheduled task.

Therefore, the migration process involves the interaction of three tasks: the migrating task, the next task on the same CPU and the next scheduled task on the other CPU. Each of these tasks performs a part of the migration:

- **The request part** is carried out by the task itself by executing the function `artis_request_for_migration()`. When the task has decided to migrate, it first sets a special flag to identify the migration step. It also sets its processor affinity to that of the only local processor in order to insure that it will not be moved unwillingly. It then re-enables the preemption and calls the scheduler. ARTiS guarantees that the task will release the CPU and that, the next time it is scheduled, it will be on the requested CPU. Then the flag and affinity are reset and the task resumes its normal course.

- **The completion part** is achieved by the "next task". When a "previous task" has set the ARTiS migration flag, it is dequeued in the scheduler, and, following the context switch, the new current task will execute the completion function `artis_complete_migration()`. It uses the special Linux callback `finish_task_switch()` which is always called after a task has finished being scheduled. The completion function chooses an NRT processor as a destination, enqueues the designated task in RT-FIFO and forces a re-schedule on the destination CPU via an inter-processor interruption.

- **The fetch part** is achieved on the destination processor. At every scheduling tick, the function `artis_fetch_migration()` is used to verify the RT-FIFOs for the NRT processors (potential migration designation). All the tasks present in those special FIFOs are pulled out and enqueued into the local run-queue.

4.1.3 Lock Free FIFO

The RT-FIFO data structure introduced in ARTiS is characterized by the fact that access to these FIFO must be lock free: RT processors should never share any lock with any NRT processor.

The algorithm proposed by Valois [10] insures that neither the pushing nor the pulling on an RT-FIFO is blocked. It is a lock free and wait free algorithm (wait free because we restrict the use of the FIFO to only one reader and one writer) based on a linked chain: one edge is pulled while another is pushed. The main characteristic of the Valois algorithm is that the list is never empty:

- on initialization, a dummy node is introduced into the structure,

- the last pulled node stays on the head list as a dummy node.

The algorithm uses nodes containing the linkage and a reference to the value (the task structure, `task_struct`, in our case). These nodes are allocated and freed dynamically.

In a real-time context, such a dynamic allocation is not affordable. The node can no longer be embedded in the task structure. This is because the node part of a pulled task would stay as dummy node in the data structure and consequently it would prevent the task being pushed again.

Our solution consists of an allocation of a node when the task structure is allocated. The node and the task structure are associated. When a task is pulled, its node stays as a dummy and the old dummy node is re-associated with the task structure.

4.2 ARTiS Load-Balancing

A load-balancing mechanism aims at optimising processor exploitation by the simple means of moving tasks from one processor to another. The aim can also be stated as being the minimization of the total running time for a given set of tasks. This is usually equivalent to maintaining the same load on every processor.

The characteristics of a load-balancer are explained in detail in [4] and can be enumerated as follows:

- information update policy: how to renew statistics on the entire system,
- trigger policy: how to decide it is time to redistribute the tasks,
- selection policy: a method for selection of imbalanced nodes,
- local designation policy: a method for selection of tasks that will be moved,
- pairing policy: a method for selection of the destination node for a given task.

The trigger policy can be either of type "pull" – under-loaded CPUs initiate the load-balancing and pull the tasks from another CPU– or "push" –over-loaded CPUs initiate the load-balancing in order to push some of their tasks– or a mix of both.

The Linux load-balancer works well, especially in real-life conditions. However with the addition of the ARTiS constraints, its behaviour is far from being optimal. In particular, the introduced asymmetry between processors requires a load-balancer that can handle the specific affinities between processors and tasks.

4.2.1 ARTiS Specific Constraints

In addition to the normal load-balancing in Linux which will only be activated inside the RT CPU set and inside the NRT CPU set, we have specified three other constraints that the load-balancer will take care of. An in-depth study of all the different load-balancing scenarii which highlights the constraints is available in [8].

Load-balancing without inter-CPU locks One of the main changes which is required from the original load-balancing mechanism is the removal of inter-CPU locks. For the same reasons as those described in section 4.1, the locks will have to be avoided in order to insure RT properties on RT CPUs.

Return of the RT1+ tasks The ARTiS migration may move RT1+ tasks from an RT CPU to an NRT CPU. However best latencies are available on the RT CPUs, so RT1+ tasks should move back to any RT CPU as soon as possible once preemption is re-enabled. Therefore, one task of the load-balancer will consist of migrating RT1+ tasks from NRT CPUs to RT CPUs.

Reduction of the RT CPUs load It might occur that an NRT CPU has less load than the RT CPUs. In this case, to get the best performance from the computer, some tasks should be migrated from an RT CPU to the NRT CPU. However, as the latencies are better on real-time processors, non real-time tasks should be given migration priority. In practice, most of the tasks trigger preemption disabling code with enough frequency so that such load-balancing is usually not needed. Still, it is necessary to handle this case in order to guarantee the best use of all the CPUs in every configuration (for instance with tasks which are only concerned with computational work).

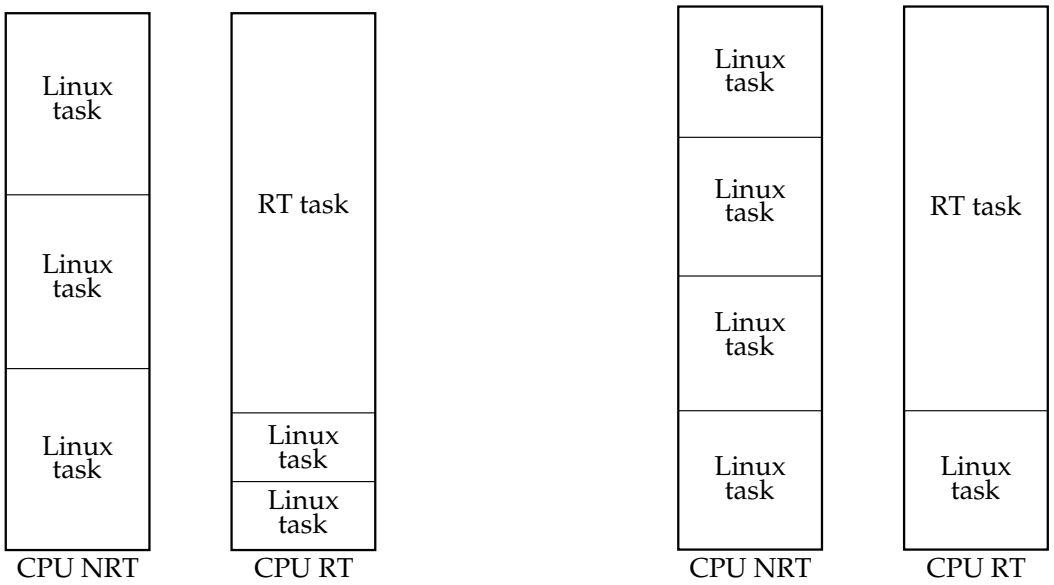
4.2.2 Load-Balancing Implementation

The current Linux implementation of load-balancing is simple, compact, modifiable and proven to work well with most of the usual workloads. Therefore, we have decided to base the load-balancer for ARTiS on this implementation. The work presented is currently in progress and is being developed on version 2.6.4 of the Linux kernel.

Run-queue length weighting The pairing policy of Linux selects the processor that will receive the tasks by locating the one which is the most loaded. The load is estimated using the number of task ready to be run (the length of the run-queue). This estimation works well as long as there are only Linux tasks being executed, this is because they share the CPU time and consequently the longer the run-queue is, the less time there is for every application.

This last assumption is false when there is a high number of real-time tasks on the computer. Because real-time tasks have an absolute priority over the other tasks, the CPU time is not shared. Therefore, the run-queue length is no longer representative of the available power. We propose improving equity between Linux tasks by adding the CPU time consumed by RT tasks as a parameter of the load estimation.

For example, on a bi-processor computer, if a real-time task consumes 3/4 of a processor time and there are 5 Linux tasks also being executed then the current Linux implementation will put 3 tasks on each processor. This implies that some Linux tasks will have 1/3 of the CPU time while others (with the same priority) will only have access to 1/8 of the time, as shown on figure 2(a). By taking into account the real consumption of the RT task, equity is recovered and every Linux task is given 1/4 of the CPU time, as shown on figure 2(b). This type of scenario is highly probable on an ARTiS system because the real-time tasks are asymmetrically distributed.



(a) *Original Mechanism* (b) *Run-queue length weighting*

FIGURE 2: *Improvement of the fairness between Linux tasks done using weighting of the run-queue length.*

The solution we propose is to measure the load of each processor using the formula $L \times \frac{1}{1-RT}$, where L is the run-queue length **without** the real-time tasks and RT is the ratio of time that was consumed by real-time tasks. Consequently, the implementation requires the addition of statistics regarding the number of RT tasks being executed on each processor, and also the measurement of the RT ratio. At a given instant the RT ratio is either 1 (there is a real-time task) or 0 (the processor is idle or executing a Linux task). To obtain the intended value it is necessary to smooth the value over time. We have chosen to use a similar mechanism to the `CALC_LOAD()` one which weights the values so that more recent values have more importance.

Inter-CPU locks withdrawal One of the direct constraint of ARTiS is avoidance of all the locks that could be taken at the same time by RT and NRT processors. Using the RT-FIFO allows this problem to be resolved but implies several changes in the load-balancer. The original version uses a "pull" trigger policy but the FIFO model is much more easily implemented within a "push" policy: a processor can just select a task, put it inside the FIFO and later on, another processor will asynchronously take it. A "pull" policy would be possible but it would be more complex and less time effective.

In order to inverse the trigger policy the main thing that needs to be changed is the function `find_busiest_queue()` which should no longer look for the longest run-queue but for the smallest one. Another implication of the change is that processors will not execute any search for a busier processor at the moment they enter into the idle state.

Next migration attempt estimation In order to provide the return of the RT1+ tasks from an NRT CPU to an RT CPU in an effective way, we had to introduce a special mechanism. Typically, an RT1+ application might call several consecutive functions that endanger real-time properties. The calls will have to be made on an NRT processor. If the load-balancer migrated it back to an RT CPU as soon as a call was finished it would lead to a ping-pong effect between the two types of processors. Not only would execution be slowed down for this task but the load-balance would not be achieved.

Therefore, we propose the modification of the task selection method so that it can favour tasks which are more likely to stay a long time on the RT processor. By simple observation of the calls endangering real-time (that is to say, a migration attempt) made by an application it is possible to obtain the frequency of the calls as well as the time of the last one. Hence, it is possible to estimate the next time a migration attempt will be made. The load-balancer can avoid migrating the tasks for which the risk of a second migration is high. This mechanism is represented in figure 4.2.2.

Typically, at a given time t, there are two possibilities:

- the next estimated migration is after t, if the migration is likely to happen soon then no mi-

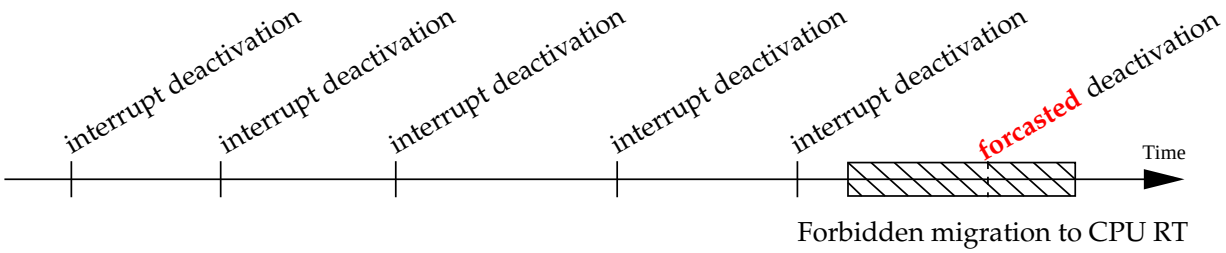


FIGURE 3: *Period of forbidden migration (hatched rectangle). The period is deducted from the study of the previous behavior of the given task.*

gration should be carried out. On the contrary, if the migration is not likely to happen for some time (specified as a system wide constant), the task can be migrated back to an RT CPU.

- the next estimated migration is before t, if it should have happened recently then it is still likely to happen soon and no migration should be carried out. If the migration was forecasted considerably beforehand, the task can be migrated back to an RT CPU. This test is relative to the measured period of the task.

A detailed mathematical representation of this conditions is available in [7].

Of course the implementation of this predicting mechanism consists in slightly modifying the load-balancer code (the function `load_balance()`) but it also consists of getting the statistics about the migration attempts. The statistics are saved inside the task structure as two numbers, one for the time weighted average period between two attempts and one for the timestamp of the last attempt. Each time the function `artis_try_to_migrate()` is called, and would trigger a migration if the current task was on an RT CPU, the statistics are updated.

Task/processor association The local designation policy (the mechanism which selects which task should be moved) and the pairing policy (the mechanism which decides the new location for a task) have to be modified so they respect the asymmetry of ARTiS. Based on the original function `load_balance()`, several versions will be derived and called depending on the types of the origin and destination CPUs.

Concerning the symmetric load-balancings (NRT to NRT and RT to RT), very little has to be done, the policies will be the same as the original one. For the load-balancing from RT to NRT, the function will be little modified so NRT tasks are moved before RT1+ tasks because the latter have better response time on the RT CPUs. Obviously, the load-balancing from NRT to RT has to behave in the opposite way by favoring the move of real-time tasks. The function will be replaced by two functions: one to specifically move the RT1+ tasks, as soon as possible, even if the RT CPU is more loaded than the original CPU. Another function will handle the move of Linux tasks but it will be executed only if the first function has not moved any task (otherwise the statistics are not correct). The two functions will also integrate the migration attempt prediction mechanism described previously.

One very important aspect of specializing the `load_balance()` function is the ability to have different triggering frequencies according to their specialization. In particular, the migration of RT1+ tasks from NRT processors to RT processors will be triggered with a high frequency. The exact frequency will be tuned during the test phase, it should be about 5 times more often than the original version so that the time the tasks spend on NRT CPUs can be minimized. It should also be noted the removal of the trigger which occurs when the processors happen to become idle, because it is not beneficial for the "push" trigger policy.

5 Conclusion and Future Work

We outlined our proposal of ARTiS, a model of real-time processing especially well suited to SMP systems. ARTiS proposes the definition of real-time tasks at the user space level and defines a basic API to do so. The main lines of the current implementation of ARTiS as a modification of the Linux kernel have been presented.

This current implementation has been evaluated on a 4 way IA-64 system [6, 5]. This evaluation measures the latencies between an IRQ waking-up a real-time task and the effective activation of the task at the user level. A maximum observed latency of 120μs has been reported on a heavily loaded system, an or-

der of magnitude less than the 40ms of the standard Linux kernel.

The ARTiS patches are available from the ARTiS web page [1] for the x86 and IA-64 architectures. Excepting small details (such as the IRQs disable mechanism), the vast majority of the code is architecture independent.

Future work covers two main aspects: the definition of interprocess communications and the definition of a scheduling for more than one hard real-time task on a processor.

ARTiS promotes a user space definition of real-time tasks, these real-time tasks must be able to communicate, typically via the shared memory or via explicit send/receive primitives. Standard communication layer implementations do not take into account either the real-time aspect or the asymmetric nature of ARTiS which distinguishes between RT and NRT processors, and which identifies different levels of RT and NRT tasks. We must propose either a new implementation, for example relying on lock-free algorithms, or new communication schemes, for example exploiting the asymmetry (real-time and non real-time tasks) between two communicating tasks.

Another limitation of the current ARTiS implementation is the definition of multiple RT0 tasks on a given processor. ARTiS allows multiple RT0 tasks on a given RT processor. These multiple RT0 tasks will not be interrupted by any other tasks. Nevertheless, it is up to the programmer to manage the share of the processor resources between these tasks. We plan to add the definition of usual real-time scheduling policies such as EDF (earliest deadline first) at this level. This extension requires

- the definition of a task model,
- the extension of the basic ARTiS API,
- the implementation of the new scheduling policies.

The new RT0 tasks would be periodic tasks running an endless loop. The ARTiS API would be extended to associate properties such as periodicity and capacity to each RT0 task. A hierarchical scheduler organization would be introduced: the current highest priority task being replaced by a scheduler that would manage the RT0 tasks.

References

[1] Laboratoire d'informatique fondamentale de Lille, Université des sciences et technologies de Lille. ARTiS home page. `http://www.lifl.fr/west/artis/`.

[2] Steve Brosky and Steve Rotolo. Shielded processors: Guaranteeing sub-millisecond response in standard Linux. In *Workshop on Parallel and Distributed Real-Time Systems, WPDRTS'03*, Nice, France, April 2003.

[3] Simon Derr and Sylvain Jeaugey. CPUSETS for Linux home page. `http://www.bullopensource.org/cpuset/`.

[4] Cyril Fonlupt. *Distribution Dynamique de Données sur Machines SIMD.* Thèse de doctorat (PhD Thesis), Laboratoire d'informatique fondamentale de Lille, Université des sciences et technologies de Lille, December 1994. (In French).

[5] ITEA Hyades Project. Linux for high performance and real-time computing on SMP systems. In *Sixth Realtime Linux Workshop*, Singapore, November 2004.

[6] Philippe Marquet, Julien Soula, Éric Piel, and Jean-Luc Dekeyser. An asymmetric model for real-time and load-balancing on Linux SMP. Research Report 2004-04, Laboratoire d'informatique fondamentale de Lille, Université des sciences et technologies de Lille, France, April 2004.

[7] Éric Piel. *Équilibrage de charge pour systèmes temps-réel asymétriques sur multi-processeurs.* Master Thesis (Mémoire de DEA), Université des sciences et technologies de Lille, Lille, France, June 2004. (In French).

[8] Éric Piel, Philippe Marquet, Julien Soula, and Jean-Luc Dekeyser. Load-balancing for a real-time system based on asymmetric multiprocessing. In *16th Euromicro Conference on Real-Time Systems, WIP session*, Catania, Italy, June 2004.

[9] Sillicon Graphics, Inc. REACT: Real-time in IRIX. Technical report, Sillicon Graphics, Inc., Mountain View, CA, 1997.

[10] John D. Valois. Implementing lock-free queues. In *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems*, Las Vegas, NV, October 1994.